X-ENS 2024 Informatique A

Éric Detrez

26 avril 2024

Arbres équilibrés et géométrie algorithmique

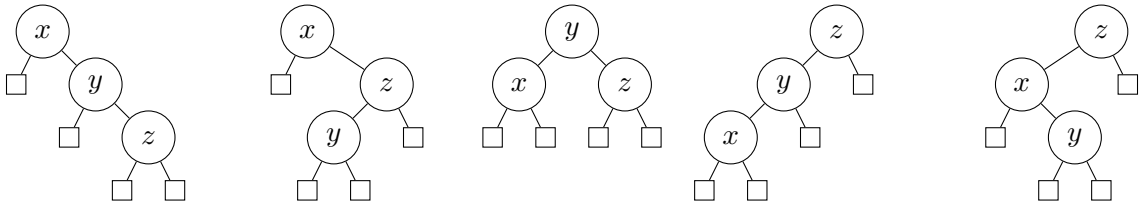
1 Préliminaires

Question 1.

Il y a deux arbres binaires de recherche contenant exactement deux éléments u et v, avec $u < v$: $\langle\langle\rangle, u, \langle\rangle\rangle, v, \langle\rangle\rangle$ et $\langle\langle\rangle, u, \langle\langle\rangle, v, \langle\rangle\rangle\rangle$.
Pour 3 éléments x, y et z, avec $x < y < z$,

- si x est à la racine, son fils gauche est vide et il y a 2 fils droits possibles avec y et z,
- si y est à la racine, son fils gauche ne contient que x et son fils droit ne contient que z,
- si z est à la racine, son fils droit est vide et il y a 2 fils gauches possibles avec x et y.

Il y a donc 5 arbres binaires de recherche contenant exactement 3 éléments.



Question 2.

Si t est un arbre binaire de recherche, son parcours infixe est croissant.
Le parcours infixe de son fils gauche est le début du parcours de t, il est donc croissant, de même le parcours du fils droit est croissant. La racine est placée après les éléments du fils gauche et avant ceux du fils droit, elle est donc supérieure aux éléments du fils gauche et inférieure aux éléments du fils droit.

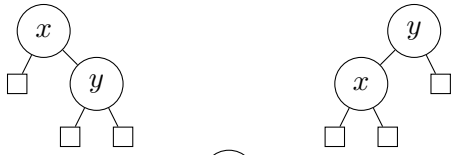
Inversement,

- si le parcours infixe du fils gauche, $(g_1, g_2, \ldots, g_p)$ est croissant, on a $g_1 < g_2 < \cdots < g_p$,
- si le parcours infixe du fils droit, $(d_1, d_2, \ldots, d_q)$ est croissant, on a $d_1 < d_2 < \cdots < d_q$,
- si la racine x est supérieure aux éléments du fils gauche et inférieure aux éléments du fils droit,on a $g_p < x < d$,

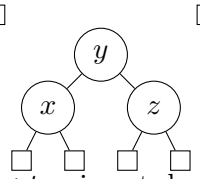
alors le parcours de t est $(g_1, g_2, \ldots, g_p, x, d_1, d_2, \ldots, d_q)$ est strictement croissant donc t est un arbre binaire de recherche.

Question 3.

Les arbres binaires de recherche à 2 éléments $x < y$ sont des arbres AVL, les fils sont de hauteur 1 et 0.

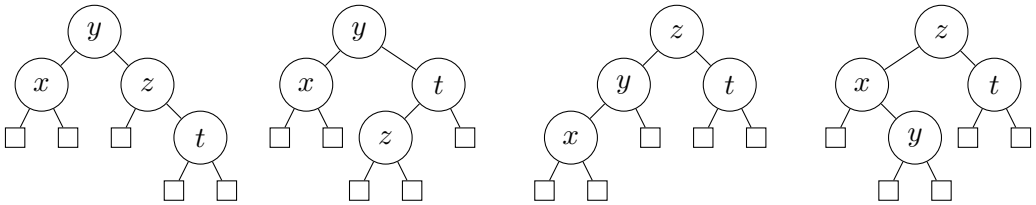


D'après la question 1, le seul arbre binaire de recherche à trois éléments $x < y < z$ qui est un arbre AVL est celui dont la racine est y.



On cherche les arbres binaires de recherche à 4, avec $x < y < z < t$ qui sont des arbres AVL.

- Si la racine est x alors le fils gauche est vide, de hauteur 0, et le fils droit a 3 éléments donc de hauteur 2 au moins. Ce ne peut pas être un arbre AVL.
- De même, si la racine est t, ce ne peut pas être un arbre AVL.
- Si la racine est y, le fils gauche n'a que x comme élément et est un arbre AVL, le fils droit admet les deux éléments z et t. Le fils droit est un arbre AVL dans les 2 cas. On trouve donc 2 arbres AVL.
- De même, si la racine est z, les deux arbres binaires de recherche possibles sont AVL.



Question 4.

On note N_h le nombre minimal d'éléments que contient un arbre AVL de hauteur h. On a $N_0 = 0$ et $N_1 = 1$. Un arbre AVL de hauteur $h+1$ contient un fils de hauteur h donc $N_{h+1} > N_h$.
Un arbre AVL de hauteur $h+2$ admet un fils de hauteur $h+1$ et un fils de hauteur h ou $h+1$ donc $N_{h+2} \geqslant \max(1 + N_{h+1} + N_h, 1 + 2 \cdot N_{h+1}) > 1 + 2 \cdot N_h$.
Si on pose $u_h = N_h + 1$ on a donc $u_{h+2} \geqslant 2 \cdot u_h$ avec $u_0 = 1$ et $u_1 = 2 \geqslant \sqrt{2}$.
Ainsi, par récurrence, $u_h \geqslant \left(\sqrt{2}\right)^h = 2^{h/2}$ d'où on déduit $\frac{h}{2} \leqslant \log(u_h)$ c'est-à-dire $h \leqslant 2 \cdot \log(N_h + 1)$. Pour un arbre t, $n(t) \geqslant N_{h(t)}$ donc $h(t) \leqslant 2 \cdot \log(n(t) + 1)$.

Question 5.

C'est la recherche classique dans un arbre binaire de recherche.

```
let rec mem x t = match t with
  | E -> false
  | N(_, _, y, _) when eq x y -> true
  | N(_, g, y, _) when lt x y -> mem x g
  | N(_, _, _, d) -> mem x d
```

Lors de chaque appel récursif la hauteur de l'arbre en paramètre diminue de 1 ou 2, il y a donc au plus $n(h)$ appels de la fonction avec, avant chaque appel au plus 2 calculs `eq` et `lt` qui s'évaluent en temps constant. La complexité est un $\mathcal{O}(h(t)) = \mathcal{O}(\log(h(t)))$

2 Construire des arbres AVL

Question 6.

La ligne 3 de `rotate_left` (resp. de `rotate_right`) est atteinte lorsque l'arbre passé en paramètre est vide ou lorsque son fils droit (resp. son fils gauche) est vide.
La ligne 12 de `join_right` est atteinte lorsque l'arbre passé en paramètre est vide.

- Lors de l'appel `join_right l x r` à la ligne 2 de `join`, on a `height l > height r +1` donc la hauteur de `l` est au moins 2, `l` est non vide et on ne parvient pas à la ligne 12 de `join_right` lors de cet appel.
- Lors de l'appel `rotate_right t` à la ligne 7 de `join_right`, on a `height t > height ll +1` donc la hauteur de `t` est au moins 2, `t` est non vide et son fils gauche, de hauteur au moins 1, est non vide : `rotate_right t` ne parvient pas à la ligne 3.
- Le résultat de `rotate_right t` n'est pas vide car il est produit par `node` donc l'argument de `rotate_left` à la ligne 7 de `join_right` est un arbre non vide avec un fils droit non vide : `rotate_left` ne parvient pas à la ligne 3.
- Lors de l'appel `join_right lr x r` à la ligne 9 de `join_right`, on a `height lr > height r +1` donc la hauteur de `lr` est au moins 2, `lr` est non vide et on ne parvient pas à la ligne 12 de `join_right` lors de cet appel.
- Lors de l'appel `rotate_left t'` à la ligne 11 de `join_right`, `t'` est non vide et sons fils droit, `t`, vérifie `height t > height ll +1` donc `t` ,est non vide : `rotate_left t'` ne parvient pas à la ligne 3.

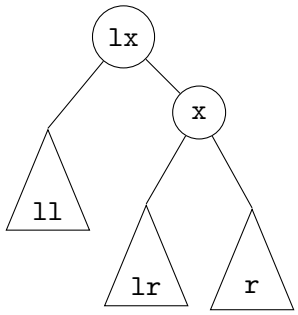
Ainsi les différents appels terminent donc `join` ne plante pas.

Question 7.

`l` est de hauteur $h \geqslant 2$ donc `l` est non vide et son fils droit `lr` est de hauteur au plus $h-1$.
On séparer les cas selon les hauteurs de `r`, $h_r \leqslant h-2$, de `ll`, $h_{ll} \in \{h-1, h-2\}$ ou de `lr`, $h_{lr} \in \{h-1, h-2\}$.

7.1 Cas $h_r = h-2$ et $h_{ll} = h-1$

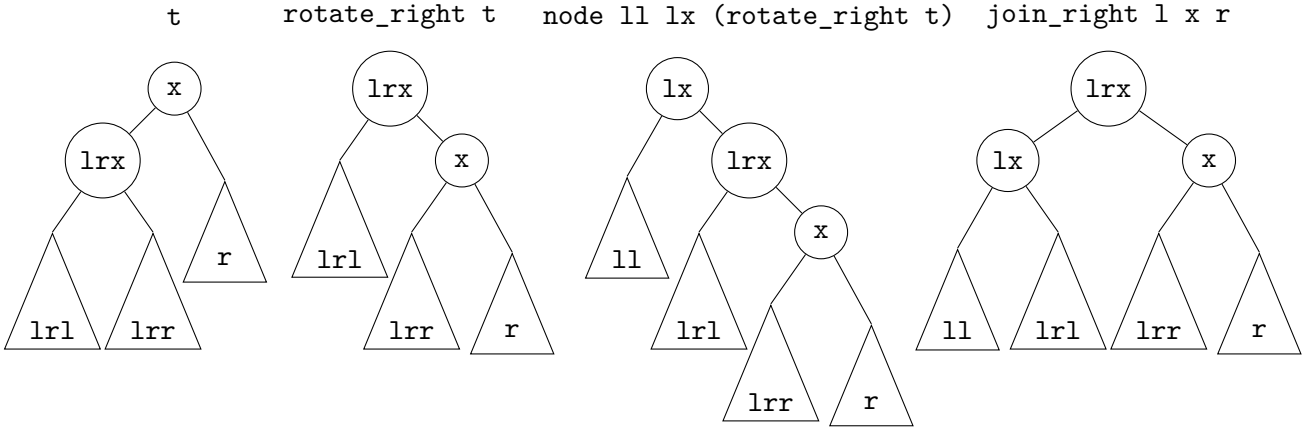
Dans ce cas la hauteur de `lr` est majorée par $h-1 = h_r$, on applique les lignes 5 à 7.
De plus `node lr x r` est de hauteur au plus $h = h_{ll}+1$, on renvoie `node ll lx t`.



Si $h_{lr} = h-1$, `t` est un arbre AVL de hauteur h et l'arbre renvoyé est un arbre AVL de hauteur $h+1$ dont le fils gauche est de hauteur $h-1$ et le fils droit est de hauteur h.
Si $h_{lr} = h-2$, `t` est un arbre AVL de hauteur $h-1$ et l'arbre renvoyé est un arbre AVL de hauteur h.

7.2 Cas $h_r = h-2$ et $h_{ll} = h-2$

Dans ce cas $h_{lr} = h-1 = h_r$, on applique encore les lignes 5 à 7. Ici `node lr x r` est de hauteur $h > h_{ll}+1$, on applique la ligne 7.
On décompose `lr` en `N(hlr, lrl, lrx, lrr)`.



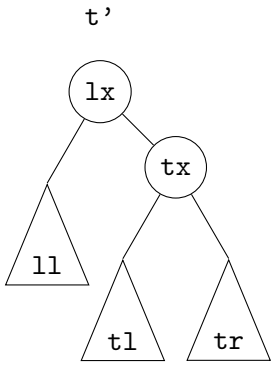
`ll` et `r` sont de hauteur $h-2$, `lrl` et `lrr` sont de hauteur $h-2$ ou $h-3$; les deux fils du résultat sont des arbres AVL de hauteur h et le résultat est un arbre AVL de hauteur h.

7.3 Cas $h_r = h-3$ et $h_{lr} = h-2$

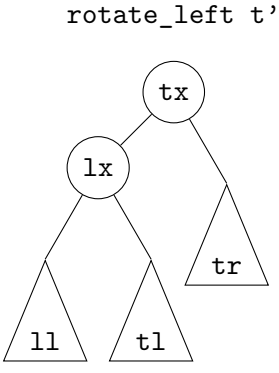
On peut appliquer de nouveau les lignes 5 à 7. Ici `node lr x r` est un arbre AVL de hauteur $h-1$ et $h_{ll} = h-1$, le résultat est donné dans la ligne 6. C'est un arbre AVL de hauteur h.

7.4 Cas $h_r = h-3$ et $h_{lr} = h-1$

L'égalité `height lr <= height r + 1` n'est plus vérifiée, on effectue les lignes 9 à 11.
Dans le calcul de `join_right lr x r` la hauteur de `r` est égale à la hauteur de `lr` diminuée de 2 et on est ramené à un des cas **7.1** ou **7.2** vus ci-dessus en remplaçant h par $h-1$. `t` est donc un arbre AVL de hauteur $h-1$ ou de hauteur h.
On décompose `t`. S'il est de hauteur h, son fils gauche, `tl`, est de hauteur $h-2$ et son fils droit, `tr`, est de hauteur $h-1$.
De plus `ll` est de hauteur $h-2$ ou $h-1$. Il y a donc 4 cas dont 2 seront regroupés.



- `t` est de hauteur h et `ll` de hauteur $h-2$. Ici `height t <= height ll + 1` n'est pas vérifiée, on effectue une rotation gauche. `ll` et `tl` sont de hauteur $h-2$ et `tr` de hauteur $h-1$ donc le résultat est un arbre AVL de hauteur h.



- `t` est de hauteur $h-1$ et `ll` de hauteur $h-2$. On renvoie `t'` ; le fils droit, `t`, est de hauteur h et le fils gauche de hauteur $h-1$ et `t'` est un arbre AVL de hauteur $h+1$.
- `t` est de hauteur h et `ll` de hauteur $h-2$ ou $h-1$. On renvoie `t'`, c'est un arbre AVL de hauteur $h+1$.

Dans tous les cas on obtient un arbre AVL de hauteur h ou de hauteur $h+1$ de fils droit de hauteur h et de fils gauche de hauteur $h-1$.

7.5 Cas $h_r < h-3$

On va procéder par récurrence sur $h - h - r$. On définit la propriété :
$\mathcal{P}(k)$: si $h_r = h_l - k$ alors `join_right l x r` renvoie un arbre AVL de hauteur h_l ou de hauteur h_l+1 de fils droit de hauteur h_l et de fils gauche de hauteur h_l-1.
On vient de prouver que $\mathcal{P}(2)$ et $\mathcal{P}(3)$ sont valides.
On suppose que $\mathcal{P}(k)$ pour tout $k \in \{2,3,\dots,p\}$, $p \geqslant 3$, et on suppose que $h_r = h_l - p - 1$.
Lors de l'appel de `join_right l x r` la hauteur du fils droit de `l` est h_l-1 ou h_l-2 et `height r + 1` est majoré par h_l-3 donc on effectue les ligne 9 à 11.
On peut appliquer l'hypothèse de récurrence pour le calcul de `join_right lr x r` car l'écart de hauteur est diminué de 1 ou 2 et on obtient un arbre AVL. On peut reproduire la démonstration du cas **7.4** qui ne fait plus intervenir `r`. On obtient bien un arbre AVL de hauteur h_l ou de hauteur h_l+1 de fils droit de hauteur h_l et de fils gauche de hauteur h_l-1.
Le résultat demandé est prouvé par récurrence.

Question 8.

Dans la question précédente, on a prouvé que si $h(l) > h(r)$, le résultat de `join l x r` est un arbre AVL de hauteur $h(l)$ ou $h(l)+1$. Par symétrie avec `join_left`, le résultat de `join l x r` est un arbre AVL de hauteur $h(r)$ ou $h(r)+1$ si $h(r) > h(l)$. Enfin, si $h(l) = h(r)$, `join l x r` renvoie un arbre AVL de hauteur $h(r)+1$.
Dans tous les cas `join l x r` renvoie un arbre AVL de hauteur h ou $h+1$ avec $h = \max\big(h(r), h(l)\big)$.

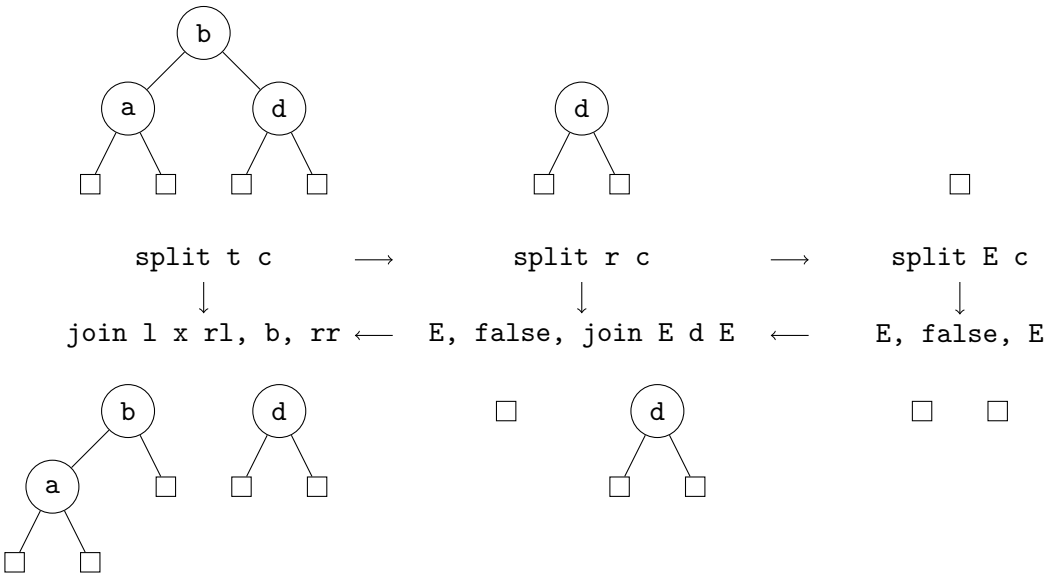
Question 9.

En dehors des appels récursifs, `join_right` ne fait qu'un nombre borné de calculs en temps constant.
Lors de chaque appel, la différence de hauteurs de `l` et `r` diminue de 1 ou 2 ; il y a au plus $h(l) - h(r)$ appels récursifs donc la complexité de `join_left` est un $\mathcal{O}\big(h(l) - h(r)\big)$.
De même la complexité de `join_right` est un $\mathcal{O}\big(h(r) - h(l)\big)$.
Comme `join` ne fait qu'un appel unique à une de ces fonctions et un nombre fini de calculs, sa complexité est bien un $\mathcal{O}\big(|h(r) - h(l)|\big)$.

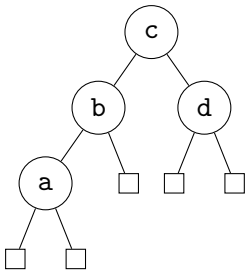
Question 10.

La fonction `split t y` découpe un arbre AVL en deux arbres AVL ; le premier contient les éléments de `t` strictement inférieurs à y et le second les éléments strictement supérieurs. Le booléen indique si y était un des éléments de `t`, dans le résultat il n'y a pas d'élement y.

Question 11.



Il reste à faire la jointure `join l c r`,



Question 12.

On peut le prouver par récurrence sur la hauteur de l'arbre AVL t.

- Si $t$est de hauteur 0, il est vide et la fonction `split` renvoie deux arbres vides qui sont des arbres AVL de hauteur 0, majorée par celle de t.
- On suppose que les arbres renvoyés par `split t y` pour de hauteur k au plus sont des arbres AVL de hauteur majorée par celle de t.
  Soit t de hauteur $k+1$. On note x sa racine et l et r ses deux fils.
  — Si $x = y$, `split t y` renvoie les arbres AVL l et r avec $h(l) < h(t)$ et $h(r) < h(t)$.
  — Si $y < x$, la fonction commence par calculer `split l y` avec l arbre AVL et $h(l) \leqslant k$. D'après l'hypothèse de récurrence, `split l y` renvoie deux arbres AVL, ll et lr, avec $h(ll) \leqslant k$ et $h(lr) \leqslant k$. `split t y` renvoie alors ll et `join lr x r` qui, d'après la question **8.**, est un arbre AVL de hauteur au plus $k+1$.
  Ainsi `split t y` renvoie deux arbres AVL de hauteur makorée par $k+1$.
  — Le cas $y > x$ est symétrique.
  La propriété pour $k+1$ est démontrée.

Question 13.

On a prouvé que la fonction `split` renvoyait deux arbres AVL, la fonction `insert` effectue, après la séparation, une fonction `join` avec des arbres AVL : elle renvoie bien un arbre AVL

La fonction `split t y`, lorsqu'elle fait un appel à elle-même, le fait avec un fils de `t`, il y a ainsi, au plus $h(t)$ appels récursifs.

À chaque itération récursive, la fonction effectue un `join` dont la complexité dépend des hauteurs des deux arbres joints ; l'un est un fils de `t` et l'autre est une des composantes de la séparation de l'autre fils de `t`. Ainsi les deux arbres ont une hauteur majorée par celle de `t` et la différence des hauteurs aussi. On aboutit donc à une complexité en $\mathcal{O}((h(t)^2))$.
Pour répondre à l'objectif de complexité en $\mathcal{O}(h(t))$, on peut remarquer que la complexité de la jointure, de l'ordre de la différence des hauteurs, n'est "grande" que si l'un des arbres joints est "petit" et que le résultat sera un "grand" arbre. On va faire intervenir les hauteurs des arbres produits dans la complexité.

On utilise des majorations plutôt que la notation $\mathcal{O}$: la comparaison $f(x) = \mathcal{O}(g(x))$ se traduit par l'existence de deux constantes K et C telles que $f(x) \leqslant K + C \cdot g(x)$ pour tout x. K est un majorant des $f(x)$ pour les x en deçà du seuil dans la définition de C.
La complexité de `join l x r` est donc majorée par $K + C \cdot (|h(r) - h(l)|)$.

Si le résultat de `split t y` est `tl, b tr`, on va montrer l'existence de 3 constantes A_1, A_2 et A_3 telles que la complexité de `split t y`, $C_{split}(t, y)$, vérifie

$$(\ddagger) \qquad C_{split}(t, y) \leqslant A_1 + A_2 \cdot h(t) + A_3 \cdot \big(h(tl) + h(tr)\big)$$

A_1, le terme constant, est choisi comme la somme des complexités de la décomposition d'un arbre en `N k l x r`, de `lt` et de `eq` et on va prouver le résultat par récurrence sur $h(t)$.

- Si t est l'arbre vide, on a bien $C_{split}(E, y) \leqslant A_1$.
- On suppose que $(\ddagger)$ est vérifiée pour tous les arbres de hauteur k au plus.
  Soit `t` de hauteur $k + 1$.
  Si on a l'égalité entre la racine de `t` et y, alors $C_{split}(t, y) \leqslant A_1$, $(\ddagger)$ reste valide.
  Sinon les fils de `t`, `l` et `r`, vérifient $k - 1 \leqslant h(l), h(r) \leqslant k$.
  On suppose que y est majoré par la racine de `t`, l'autre cas est symétrique.
  On a avec les notations du code, $C_{split}(t, y) \leqslant A_1 + C_{split}(l, y) + C_{join}(lr, x, r)$.
  D'après l'hypothèse de récurrence, $C_{split}(l, y) \leqslant A_1 + A_2 \cdot h(l) + A_3 \cdot \big(h(ll) + h(lr)\big)$.
  On a $h(l) \leqslant h(t) - 1$ et `tl = ll`. donc $C_{split}(l, y) \leqslant A_1 + A_2 \cdot h(t) - A_2 + A_3 \cdot h(tl) + A_3 \cdot h(lr)$
  — On commence par le cas où $h(lr) \leqslant h(r)$. D'après les question **7.** et **8.** la taille de `tr = join lr x r` est comprise entre $h(r)$ et $h(r) + 1$ d'où $h(r) \leqslant h(tr)$.
  Si on choisit $A_3 = C$ on obtient $C_{join}(t, y) \leqslant K + A_3 \cdot h(tr) - A_3 \cdot h(lr)$.
  — On a toujours $h(lr) \leqslant h(l) \leqslant h(r) - 1$ donc on ne peut avoir $h(lr) > h(r)$ que dans le cas où $h(lr) = h(l) = h(t) - 1$ et $h(r) = h(t)$. Dans ce cas la fonction `join` n'effectue qu'un appel à `node` pour produire un arbre de hauteur $h(lr) + 1 = h(tr)$.
  Ainsi $1 = h(tr) - h(lr)$ donc $C_{join}(t, y) \leqslant K + A_3 = K + A_3 \cdot h(tr) - A_3 \cdot h(lr)$
  Les deux cas donnent la même inégalité qui donne ensuite
  $C_{split}(t, y) \leqslant A_1 + A_1 + A_2 \cdot h(t) - A_2 + A_3 \cdot h(tl) + A_3 \cdot h(lr) + K + A_3 \cdot h(tr) - A_3 \cdot h(lr)$.
  Il suffit de choisir $A_2 = A_1 + K$ pour obtenir
  $C_{split}(t, y) \leqslant A_1 + A_2 \cdot h(t) + A_3 \cdot h(tl) + A_3 \cdot h(tr)$.

La récurrence est prouvée.

On en déduit que `split t y` a une complexité en $\mathcal{O}(h(t))$.
Il est alors de même pour `insert x t` et la majoration de la question **4.** donne bien une complexité en $O(\log(n(t)))$.

Question 14.

Il y a i entier stockés dans l'arbre `a.(i)` donc au total $\displaystyle\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = f(n)$ entiers.
Pour construire `a.(i)` à partir de `a.(i-1)` on effectue une insertion sur un arbre de taille $i - 1$; comme la complexité est en $O(\log(i - 1))$, on a fait au plus $K + C \cdot \log(i - 1)$ appels à `node`.

On a donc au plus $K + \sum_{i=2}^{n-1} C \cdot \log(i-1) \leqslant K \cdot n + C \cdot n \cdot \log(n) = \mathcal{O}(n \cdot \log(n))$ nœuds en tout, ce qui est un $o(f(n))$.

Question 15.

```
let rec split_last t = match t with
  | E -> failwith "L'arbre est vide"
  | N (_, l, x, E) -> l, x
  | N (_, l, x, r) -> let r', y = split_last r in join l x r
   ', y
```

Question 16.

```
let join2 l r =
  let l', x = split_last l in join l' x r
```

Question 17.

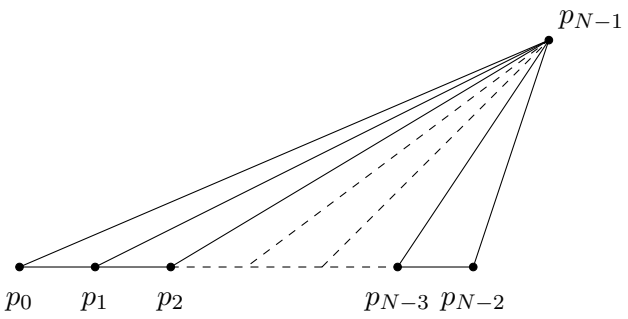
```
let remove x t =
  let l, _, r = split x t in join2 l r
```

8 Application : localisation d'un point dans le plan

Au sujet de cette partie, on pourra écouter ce cours de Xavier Leroy [1] au collège de France.
La présentation de l'algorithme commence à partir de 32 minutes 30.
On interprète la phrase "*Les arêtes ne se croisent pas*" par le fait que l'intersection de deux arêtes ne peut être qu'une extrémité commune.

Question 18.

On peut choisir $N-1$ points p_i de coordonnées $(i, 0)$ pour $0 \leqslant i \leqslant N-2$ et p_{n-1} de coordonnées $(N-1, N-1)$ avec les $N-2$ polygones (p_i, p_{i+1}, n_{N-1}). On a alors $T_0 = 0, T_1 = 2$, $T_2 = 3$, …, $T_{N-2} = T_{N-1} = N-1$ et $T_N = 0$ d'où un total de $\frac{N^2+N-4}{2} = \theta(N^2)$.

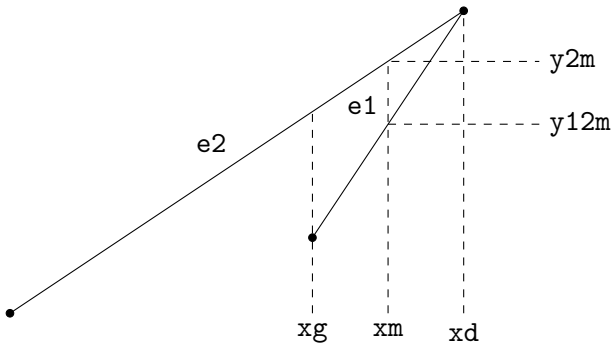


1. https ://www.college-de-france.fr/fr/agenda/cours/structures-de-donnees-persistantes/rien-ne-se-perd-tout-se-cree-introduction-aux-structures-de-donnees-persistantes

Question 19.

Les deux arêtes traversent une tranche commune, elles peuvent traverser plusieurs tranches qui sont alors contiguës. On calcule les abscisses des extrémités de l'union des tranches communes aux deux arêtes, `xg` et `xd`.
On calcule alors les ordonnées, `y1m` et `y2m`, des deux arêtes en `xm`, milieu de `xg` et `xd`. Comme les deux arêtes ne peuvent se rencontrer qu'en leurs extrémités on doit avoir `y1m <> y2m` donc l'arête au dessus de l'autre est celle pour laquelle cette ordonnée est la plus grande.

```
let lt e1 e2 =
  let (x1g, y1g), (x1d, y1d) = e1 in
  let (x2g, y2d), (x2d, y2d) = e2 in
  let xg = if x1g < x2g then x2g else x1g in
  let xd = if x1d < x2d then x1d else x2d in
  let xm = (xg +. xd) /. 2 in
  let y1m = y1g +. (y1d -. y1g)*.(xm -. x1g)/.(x1d - x1g) in
  let y2m = y2g +. (y2d -. y2g)*.(xm -. x2g)/.(x2d - x2g) in
  y1m < y2m;
```



Question 20.

Le nombre d'arêtes est N' ; on a admis que $N' = \mathcal{O}(N)$.

1. On crée le tableau `extr` des $2N'$ abscisses des extrémités..... $\mathcal{O}(N)$ Chaque abscisse est accompagnée de l'arête qui la définit.
2. On trie le tableau `extr` ..... $\mathcal{O}(N \cdot \log(N))$
3. On compte le nombre d'éléments distincts, N ..... $\mathcal{O}(N)$
4. On crée le tableau `slabs` de taille $N + 1$ avec des arbres vides en choisissant les valeurs distinctes dans `extr` pour les `xleft` et `xright` ..... $\mathcal{O}(N)$
5. On définit $i = 0$.
6. Pour chaque abscisse dans `extr` ..... $2 \cdot N'$ itérations
   (a) si elle est la valeur de `slabs.(i).xright`, on ajoute ou retire selon le cas l'arête associée à `slabs.(i).tree` et on place le résultat dans `slabs.(i+1).tree` puis on incrémente i ..... $\mathcal{O}(\log(N))$
   (b) sinon, on ajoute ou retire selon le cas l'arête associée à `slabs.(i).tree` et on place le résultat dans `slabs.(i).tree` ..... $\mathcal{O}(\log(N))$

   Pour toute la boucle ..... $\mathcal{O}(N \cdot \log(N))$

La complexité totale est donc en $\mathcal{O}(N \cdot \log(N))$.

Question 21.

Dans la fonction `chercher_entre i j`, `x` est compris entre `tranches.(i).xleft` et `tranches.(j).xright`

```
let find_slab tranches x =
  let rec chercher_entre i j =
    if i = j
    then i
    else let m = (i + j)/2 in
      if x > tranches.(m).xright
      then chercher_entre (m+1) j
      else chercher_entre i m in
  let n = Array.length tranches in
  chercher_entre 0 (n-1)
```

Question 22.

On introduit un type pour avoir la possibilité d'une arêtes infiniment basse ou infiniment haute :

```
type arete = Out | Ar of edge
```

On définit une fonction qui teste si un point est au-dessus d'une arête

```
let dessus p e =
  let x, y = p
  let (xg, yg), (xd, yd) = e in
  let t = (x -. xg) /. (xd -. xg) in
  let ye = yg +. t *. (yd -. yg) in
  ye < y
```

On va ensuite chercher dans l'arbre récursivement en encadrant par des arêtes en-dessous et au dessus, `out` est toujours en dessous comme première arête et au-dessus comme seconde arête.

```
let find tranches p =
  let x, y = p in
  let k = find_slab tranches x in
  let rec chercher_dans t a1 a2 in
    match t with
    | E -> a1, a2
    | N(_, l, e, r) -> if dessus p e
      then chercher_dans r (Ar e) a2
      else chercher_dans l a1 (Ar e) in
  match chercher_entre Out Out tranches.(k).tree with
  | Out, _
  | _, Out -> Outside
  | Ar e1, Ar e2 -> Between (e1, e2)
```

Ce travail contient certainement des erreurs et des imprécisions. Merci de les signaler à info[at]eric-detrez[dot]fr